TPLP **21** (6): 700–716, 2021. © The Author(s), 2021. Published by Cambridge University Press
doi:10.1017/S1471068421000272 First published online 23 September 2021

Transformation-Enabled Precondition Inference

BISHOKSAN KAFLE
IMDEA Software Institute, Madrid, Spain
(*e-mail:* bishoksan.kafle@imdea.org)

GRAEME GANGE and PETER J. STUCKEY
Faculty of IT, Monash University, Clayton Vic. 3800, Australia
(*e-mails:* graeme.gange@monash.edu, peter.stuckey@monash.edu)

PETER SCHACHTE and HARALD SØNDERGAARD
School of Computing and Information Systems,
The University of Melbourne, Vic. 3010, Australia
(*e-mails:* schachte@unimelb.edu.au, harald@unimelb.edu.au)

submitted 06 August 2021; accepted 22 August 2021

Abstract

Precondition inference is a non-trivial problem with important applications in program analysis and verification. We present a novel iterative method for automatically deriving preconditions for the safety and unsafety of programs. Each iteration maintains over-approximations of the set of *safe* and *unsafe initial* states, which are used to partition the program's *initial* states into those known to be *safe*, known to be *unsafe* and *unknown*. We then construct revised programs with those *unknown* initial states and iterate the procedure until the approximations are disjoint or some termination criteria are met. An experimental evaluation of the method on a set of software verification benchmarks shows that it can infer precise preconditions (sometimes optimal) that are not possible using previous methods.

KEYWORDS: precondition inference, program transformation, abstract interpretation, optimal precondition.

1 Introduction

Precondition analysis infers input conditions that establish runtime properties of interest (e.g., a *sufficient precondition* for safety is a set of initial states, each of which is guaranteed to be safe with respect to given safety properties). Applications include program verification, symbolic execution, debugging, and program comprehension. Derivation of exact preconditions (excluding no good runs and including no bad runs) is undecidable, so the aim is to derive preconditions that are general enough to be useful in practice. We approach the problem by iteratively refining over-approximations of safe and unsafe states. For this, constrained Horn clauses (CHCs) are convenient, as they can model imperative programs and assertions in a uniform way (Peralta *et al*. 1998; Grebenshchikov *et al*. 2012; Gurfinkel *et al*. 2015; De Angelis *et al*. 2017).

Consider the program in Figure 1. The left box shows a fragment in C, the right box shows its CHC representation, encoding reachable states. C variables are represented by logical variables (capital letters). The clause c_1 specifies the initial states of the program

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

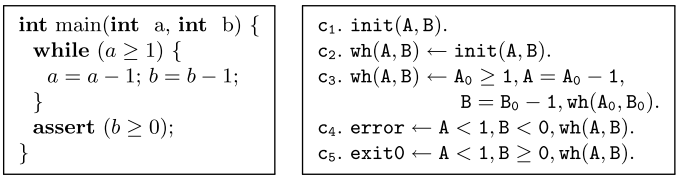


Fig. 1. Running example: (left) original program, (right) translation to CHCs.

via the predicate *init* which is always reachable. Similarly, c_2 and c_3 encode the reachability of the *while* loop via the predicate *wh*. Clause c_2 states that the loop is reachable if *init* is reachable while c_3 states that the loop is (re-)reachable from the end of its own body (recursive case). The last two clauses represent the properties of the program. Clause c_4 states that an "unsafe or error" state is reached if $\mathtt{B} < 0$ upon loop exit (encoded by `error`), and the clause c_5 states that the program terminates gracefully or reaches a safe state if $\mathtt{B} \geq 0$ (encoded by `exit0`). The program is *unsafe* if it reaches `error` for some input and is *safe* if none of the input reaches `error`. (The semantics of *assert(c)* is *if(c)* `SKIP` *else* `ERROR`.)

Clearly the program terminates. Its assertion is violated if the initial conditions on `a` and `b` entail the disjunction $(\mathtt{b} < 0 \wedge \mathtt{a} \leq 0) \vee (\mathtt{a} \geq 1 \wedge \mathtt{a} > \mathtt{b})$. It terminates gracefully if $(\mathtt{b} \geq 0 \wedge \mathtt{a} \leq 0) \vee (\mathtt{a} \geq 1 \wedge \mathtt{b} \geq \mathtt{a})$. Automatic derivation of these preconditions is challenging for at least three reasons:

(i) The desired result is a disjunction of linear constraints – to reach it, we need the ability to express disjunctive information.

(ii) Invoking an abstract interpreter using forward analysis on the original program derives $a \geq 1$ as invariant for predicate `wh`, while an abstract interpreter working backward from a goal such as `exit0` derives $b \geq 1$ as invariant for the same predicate. That is, without a more sophisticated approach, we lose critical information about `a` and `b` in backward and forward analysis, respectively.

(iii) We need to reason simultaneously about the safe and unsafe states; one type of information cannot simply be obtained by complementing the other. For example, the formula $(\mathtt{b} \geq 0 \wedge \mathtt{a} \leq 0) \vee (\mathtt{a} \geq 1 \wedge \mathtt{b} \geq \mathtt{a})$ that implies safe termination cannot be obtained by negating the formula $(\mathtt{b} < 0 \wedge \mathtt{a} \leq 0) \vee (\mathtt{a} \geq 1 \wedge \mathtt{a} > \mathtt{b})$ that implies violation of the assertion. Previous approaches (Howe *et al*. 2004; Moy 2008; Miné 2012; Bakhirkin *et al*. 2014; Kafle *et al*. 2018) fail to infer the desired preconditions, either because they can only infer conjunctive information, or because they rely on an abstract operation of complementation that comes with some loss of precision.

The method we present addresses the challenges as follows. Challenge (i) is addressed via partial evaluation that creates a finite number of versions of each predicate, which is essential for deriving disjunctive invariants. Many loops in typical program patterns, such as our example, require disjunctive invariants to be established. While there are techniques that are capable of inferring precise invariants (Sankaranarayanan *et al*. 2004; Gulwani *et al*. 2009; Gupta and Rybalchenko 2009; Beyer *et al*. 2007), we argue that they are significantly more complicated, less efficient, and less widely used than standard abstract interpretation-based techniques for generating conjunctive invariants (Karr 1976; Cousot and Halbwachs 1978; Miné 2006). In this article we use the standard invariant generation tools in conjunction with program transformations with the aim of raising their preci-

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

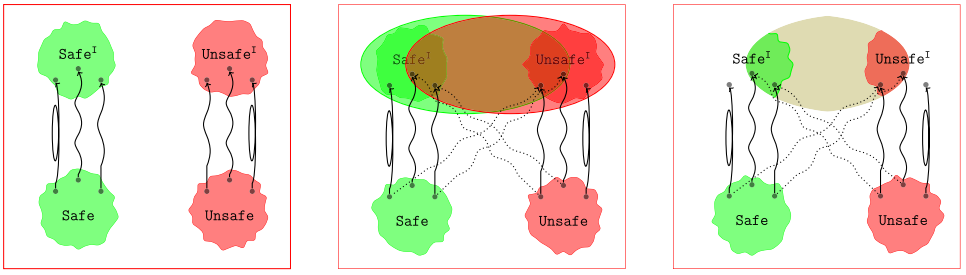


Fig. 2. Precondition inference: Reality (left), initial approximations (middle), one step refinement of approximations using Algorithm 1 (right). Arrows represent preconditions flowing from safe or unsafe final states (bottom) back to corresponding initial states (top).

sion level to that of disjunctive invariant generation tools. Challenge (ii) is addressed via forward and backward abstract interpretation, together with constraint specialization. Using this, one can infer $a \geq 1, b \geq a$ as invariant for predicate `wh`, as described in Section 3. Challenge (iii) is addressed by concurrently maintaining and refining approximations of both safe and unsafe states, until the approximations are disjoint or some termination criterion is met.

Figure 2 sketches the idea. The leftmost panel reflects reality, showing the set of concrete *safe* and *unsafe* states, along with the corresponding sets of *initial* states. Given a program and a description of the sets of interest (safe and unsafe states), precondition analysis infers a set of initial states that lead to these sets of interest. Or rather, it finds *over-approximations* of the initial sets of states – so these may overlap. Over-approximations are shown as ellipses in the middle panel, colored appropriately for safe and unsafe initial states. Because of approximation, there may be witness traces from the left ellipse to `Unsafe` (the dotted arrows) and vice versa. The algorithm aims to reduce the ellipses progressively to the point where the ellipses no longer overlap. A single refinement step that focuses our attention only on the intersection is illustrated in the rightmost panel.

Our work builds upon the transformation-guided framework of Kafle *et al.* (2018) and incorporates a number of program transformations known from the literature, including

- Partial evaluation (Jones *et al.* 1993): PE wrt. a goal specializes a program for the given goal; preserving only those derivations that are relevant for deriving the goal.
- Constraint Specialization (Kafle and Gallagher 2017a) via forward and backward abstract interpretation (Bakhirkin and Monniaux 2017): This strengthens constraints in clauses by exploiting generated invariants, while preserving derivations of a goal. The effect is to prune paths that are not relevant for deriving the goal.
- Trace Elimination (Kafle and Gallagher 2017b): This eliminates a set of traces from a program while preserving the rest of traces and serves to refine a program.

Our contribution is to combine these techniques into an iterative framework that can control the quality of the preconditions for both safety and unsafety. Kafle *et al.* (2018) iteratively apply CHC transformations to a program wrt. error, approximating the unsafe states whose complement yields sufficient preconditions for the safety. A disadvantage of this is the *blind* refinement of unsafe states without knowing its frontier with the safe states. This misses opportunities to avoid redundant computation as well as to guide the refinement process at an early stage. We extend that work in a number of directions:

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

- We model both the safe and unsafe program states, and refine them concurrently, allowing us to derive preconditions for both safety and unsafety. In addition, we show how to derive preconditions for both safety and unsafety either from the original program or one obtained via a sequence of transformations (Section 3).
- We present an iterative algorithm to refine approximations of these states, each iteration focusing only on states yet to be shown safe or unsafe (the intersection of safe and unsafe over-approximations), thus reducing the search space. It has refined termination criteria to control precision and detect optimality of the preconditions.
- Reasoning simultaneously about safe and unsafe states allows us to derive precondition for non-termination as a complement of necessary preconditions for safety and unsafety (Section 4).
- Evaluation shows that we not only infer non-trivial preconditions in slightly more cases but also infer optimal ones in some cases (Section 5).

2 Preliminaries

An *atom* is a formula $p(\mathbf{x})$ where p is a predicate symbol and $\mathbf{x}$ a tuple of arguments. A CHC is a first-order formula written as $p_0(\mathbf{x}_0) \leftarrow \varphi, p_1(\mathbf{x}_1), \ldots, p_k(\mathbf{x}_k)$ following Constraint Logic Programming (CLP) standard, where φ is a finite conjunction of quantifier-free *constraints* on variables $\mathbf{x}_i$ with respect to some constraint theory $\mathbb{T}$, $p_i(\mathbf{x}_i)$ are atoms. A *constrained fact* is a clause of the form $p_0(\mathbf{x}_0) \leftarrow \varphi$, where φ is a constraint. We assume the theory $\mathbb{T}$ is equipped with a decision procedure and a projection operator, and that it is closed under negation.

The notation $\varphi|_V$ represents the constraint formula φ projected onto variable set V and $\varphi \models_{\mathbb{T}} \psi$ (or equivalently $\models_{\mathbb{T}} \varphi \rightarrow \psi$) to represent φ entails ψ over $\mathbb{T}$. Similarly, we write $P \vdash_{\mathbb{T}} A$ when an atom A is derivable from the program P wrt. the theory $\mathbb{T}$.

We use CHCs to encode control flow of C-like programs. Two special predicates, exit0 and error, encode safe and unsafe (error) states, respectively. So exit0 indicates a normal return; error indicates abnormal termination. The predicate init encodes the set of initial states. We assume users specify all states of interest by appropriate constructs provided by the language (e.g. *assert(c), return* $\langle n \rangle$ of C). States not specified by the user (e.g. *buffer overflow, floating point exceptions*) are not taken into account while generating CHCs. Hence correctness of the preconditions depends on the user specified set of states.

From here on, when talking about a program, we refer to its CHC representation.

Definition 1 (*AND-tree (Gallagher and Lafave 1996)*)
An *AND-tree* for a CHC program P is a tree whose nodes are labeled as follows.

1. Each non-leaf node corresponds to a (renamed) clause in P of the form $A \leftarrow \varphi, A_1, \ldots, A_k$ where $k > 0$. The clause is renamed so that any variables not appearing in A are fresh. The node is labeled by (A, φ). The node has k child nodes where the i^{th} child corresponds to a clause in P of the form $A_i \leftarrow \varphi_i, B_i$ where B_i may be empty and is labeled by (A_i, φ_i).
2. Each leaf node corresponds to a (renamed) clause in P of the form $A \leftarrow \varphi$ and is labeled as (A, φ).

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

Given an AND-tree t, $\mathsf{constr}(t)$ is the conjunction of the constraints appearing in the tree. The tree t is *feasible* if and only if $\mathsf{constr}(t)$ is satisfiable over $\mathbb{T}$.

Definition 2 (*Initial clauses and nodes*)
Let P be a program with a distinguished predicate p^I which we call the *initial predicate*. The *constrained facts* $\{(p^I(\mathbf{x}) \leftarrow \theta) \mid (p^I(\mathbf{x}) \leftarrow \theta) \in P\}$ are the *initial clauses* of P. Let t be an AND-tree for P. A node labeled by $p^I(\mathbf{x}) \leftarrow \theta$ is an *initial node* of t. We extend the term "initial predicate" and use the symbol p^I to refer also to renamed versions of the initial predicate that arise during clause transformations.

In Figure 1 the initial predicate is `init` and the initial clause is `init(A,B)`.

3 Program transformations and preconditions

We now show how to find preconditions for safety and unsafety for a program (original or obtained via transformation). We limit attention to sets of clauses for which every AND-tree for exit0 and error (whether feasible or infeasible) has at least one initial node.

Definition 3 (*Program with initial states φ (P^I_φ) and replaced states φ (P^R_φ)*)
Let P be a program and φ a constraint over $\mathbb{T}$. Let P^I_φ be the clauses obtained from P by replacing the initial clauses $\{(p^I(\mathbf{x}) \leftarrow \theta_i) \mid 1 \leq i \leq k\}$ by $\{(p^I(\mathbf{x}) \leftarrow \varphi \wedge \theta_i) \mid 1 \leq i \leq k\}$. Similarly, let P^R_φ be the set of clauses obtained from P by replacing the initial clauses $\{(p^I(\mathbf{x}) \leftarrow \theta_i) \mid 1 \leq i \leq k\}$ by $\{(p^I(\mathbf{x}) \leftarrow \varphi)\}$.

Definition 4 (*Necessary/sufficient precondition for safety*)
Let P be a program and φ a constraint over $\mathbb{T}$. Then

- a constraint φ is a *necessary precondition* (NP) for the safety of P if $P \vdash_{\mathbb{T}} \mathsf{exit0}$ entails $P^R_\varphi \vdash_{\mathbb{T}} \mathsf{exit0}$. In words, φ (possibly *true*) is an over-approximation of the set of initial states of P that can reach exit0.
- a constraint ψ is a *sufficient precondition* (SP) for the safety of P if $P^I_\psi \not\vdash_{\mathbb{T}} \mathsf{error}$. In words, ψ (possibly *false*) is an under-approximation of the set of initial states of P that cannot reach error.

Thus an SP for safety is a constraint that suffices to block derivations of error (given we assume clauses for which p^I is essential for any derivation of error). In practice we would like to consider SP for safety as a constraint that allows derivations of exit0 *and* blocks derivations of error. We define NP ($\mathsf{nec_u}(P)$) and SP ($\mathsf{suf_u}(P)$) for unsafety analogously. In the following, we show how an NP and an SP can be derived from a set of clauses.

Definition 5 (*NP extracted from CHC program P*)
Let P be a set of clauses encoding reachable states of a program. The formula

$$\bigvee \{\theta \mid (p^I(\mathbf{x}) \leftarrow \theta) \in P\}.$$

is an NP for *both* safety and unsafety. We refer to it as $\mathsf{nec_s}(P)$ when talking about safety and as $\mathsf{nec_u}(P)$ when talking about unsafety.

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

The reason why $\mathsf{nec_s}(P)$ is an NP for safety is that any feasible AND-tree for P must use at least one initial clause of P and so the disjunction of constraints from the initial clauses (although imprecise) is a sound NP for the safety, as well as the unsafety, of P. Using this definition, the NPs for the program P in Figure 1 are $\mathsf{nec_s}(P) = \mathsf{nec_u}(P) = \mathit{true}$ (the set of initial states of the program). Given NPs for a program P, we can find *sufficient preconditions* for the safety and unsafety as follows.

Definition 6 (*SP extracted from CHC program P*)
Let P be a set of clauses encoding reachable states of a program. We define

$$\begin{array}{lcl} \mathsf{suf_s}(P) & = & \mathsf{nec_s}(P) \wedge \neg \mathsf{nec_u}(P) \\ \mathsf{suf_u}(P) & = & \mathsf{nec_u}(P) \wedge \neg \mathsf{nec_s}(P) \end{array}$$

The former is a sufficient precondition for safety, the latter for unsafety, of P. In the sequel, we represent necessary and sufficient preconditions by Greek letters φ and ψ, respectively, possibly with subscript u for unsafety and s for safety.

Let φ_c be $\mathsf{nec_s}(P) \wedge \mathsf{nec_u}(P)$, denoting the shared region between the (approximate) safe and unsafe states. A precondition is *separating* (optimal) if φ_c is *unsatisfiable* (the safe and unsafe regions are separated). Then we have $\mathsf{suf_s}(P) = \mathsf{nec_s}(P)$ since $\mathsf{nec_s}(P) \models_{\mathbb{T}} \neg\mathsf{nec_u}(P)$. That is, necessary and sufficient conditions are the same for the validity of the assertion. Analogously, we have $\mathsf{suf_u}(P) = \mathsf{nec_u}(P)$.

The shared region characterized by φ_c indicates imprecision of over-approximations – which we attempt to reduce as much as possible. We achieve this reduction as follows:

1. Construct a revised program $P^I_{\varphi_c}$ (Definition 3) from P focusing only on the shared region such that its SP is a valid SP for P (Lemma 3.1).
2. Shrink either of the regions via iterative strengthening of the initial clauses of the program from where necessary preconditions are derived.

For this we utilize well-known CHC transformations, from the literature on CLP and Horn clause verification, as outlined below.

Proposition 3.1
Let P be a program, φ a constraint and P^I_φ as defined in Definition 3. Let ψ be any SP for the safety (unsafety) of P^I_φ. Then ψ is also an SP for the safety (unsafety) of P.

1. Partial Evaluation (PE). The PE algorithm we use (Gallagher 2019) produces a polyvariant specialization, that is, a finite number of versions of each predicate, which is essential for deriving disjunctive information as well as for refining the control flow of the program (Doménech *et al.* 2019). The result of applying PE to the example program in Figure 1 wrt. `error` and `exit0` is shown in Figure 3. For details we refer to Gallagher (2019). A key point is that, owing to polyvariant specialization, `init` and `wh` are split into two different versions, leading to more precise preconditions as in equation (1) (using Definition 5).

$$\begin{array}{l} \varphi_u = (\mathtt{B} < 0 \wedge \mathtt{A} \leq 0) \vee \mathtt{A} \geq 1 \equiv \mathtt{B} < 0 \vee \mathtt{A} \geq 1 \\ \varphi_s = (\mathtt{B} \geq 0 \wedge \mathtt{A} \leq 0) \vee (\mathtt{A} \geq 1 \wedge \mathtt{B} \geq 0) \equiv \mathtt{B} \geq 0 \end{array} . \tag{1}$$

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

```
error ← B < 0, A ≤ 0, wh_2(A, B).
wh_2(A, B) ← B < 0, A ≤ 0, init_2(A, B).
wh_2(A, B) ← B < 0, A = 0, C = 1,
                B − D = −1, wh_1(C, D).
wh_1(A, B) ← A ≥ 1, init_1(A, B).
wh_1(A, B) ← A ≥ 1, A − C = −1,
                B − D = −1, wh_1(C, D).
init_1(A, B) ← A ≥ 1.
init_2(A, B) ← A ≤ 0, B < 0.
```

```
exit0 ← B ≥ 0, A ≤ 0, wh_2(A, B).
wh_2(A, B) ← B ≥ 0, A ≤ 0, init_2(A, B).
wh_2(A, B) ← B ≥ 0, A = 0, C = 1,
                B − D = −1, wh_1(C, D).
wh_1(A, B) ← A ≥ 1, B ≥ 0, init_1(A, B).
wh_1(A, B) ← B ≥ 0, A ≥ 1, A − C = −1,
                B − D = −1, wh_1(C, D).
init_1(A, B) ← A ≥ 1, B ≥ 0.
init_2(A, B) ← B ≥ 0, A ≤ 0.
```

Fig. 3. Partially evaluated programs: wrt. `error` (left) and wrt. `exit0` (right).

```
wh_1(A, B) ← A > B, A ≥ 1, init_1(A, B).
wh_1(A, B) ← A > B, A ≥ 1, A − C = −1,
                B − D = −1, wh_1(C, D).
init_1(A, B) ← A > B, A ≥ 1.
init_2(A, B) ← B < 0, A ≤ 0.
```

```
wh_1(A, B) ← A ≥ 1, B ≥ A, init_1(A, B).
wh_1(A, B) ← B ≥ A, A ≥ 1, A − C = −1,
                B − D = −1, wh_1(C, D).
init_1(A, B) ← A ≥ 1, B ≥ A.
init_2(A, B) ← B ≥ 0, A ≤ 0.
```

Fig. 4. Constraint specialized programs: wrt. `error` (left) and wrt. `exit0` (right).

2. Constraint Specialisation (CS). CS (Kafle and Gallagher 2017a) of program P wrt. goal A specializes each constraint φ in a clause of P to a constraint $\varphi \wedge \psi$ while preserving the derivation of A. Figure 4 shows application of CS to Figure 3 (left) wrt. error, and to Figure 3 (right) wrt. exit0. Note that only clauses that got specialized are shown, and the newly derived constraints are underlined for readability. The underlined constraints in Figure 4 (left) are obtained by recursively propagating $\mathtt{B} < 0, \mathtt{A} \leq 0$ top-down from the goal `error` and $\mathtt{A} \geq 1$ bottom-up from the initial clause using program transformation and abstract interpretation over the domain of convex polyhedra. In more detail, we first compute the query–answer transformed version (Codish and Søndergaard 2002) of the program in Figure 3 (left) wrt. the goal `error` (thus simulating the top-down computation), then apply abstract interpretation. An excerpt of the query–answer transformed program (just enough to show the provenience of the constraint $\mathtt{A} > \mathtt{B}$) is shown in Figure 5.

Since `wh_1_q` is the only recursive predicate, the rest can be unfolded away, leaving two clauses: $\mathtt{wh_1_q}(\mathtt{A}, \mathtt{B}) \leftarrow \mathtt{A} = 1, \mathtt{B} \leq 0$ and $\mathtt{wh_1_q}(\mathtt{C} + 1, \mathtt{D} + 1) \leftarrow \mathtt{wh_1_q}(\mathtt{C}, \mathtt{D}), \mathtt{C} \geq 1$. Abstract interpretation using the polyhedral domain derives $\mathtt{A} \geq 1, \mathtt{A} > \mathtt{B}$ as invariant for $\mathtt{wh_1_q}(\mathtt{A}, \mathtt{B})$ since we have the constraint $\mathtt{A} = 1, \mathtt{B} \leq 0$ in the base case, while A and B are incremented in lockstep in the recursive case.

The computed invariant $\mathtt{A} > \mathtt{B}$ for `wh_1_q(A,B)` in derivations of error is conjoined to each call of `wh_1`, since the invariant holds in each such call. The underlined constraint $\mathtt{B} \geq \mathtt{A}$ in Figure 4 (right) is obtained in similar way. Using these specialized programs, we derive the necessary preconditions:

$$\begin{array}{l} \varphi_u = (\mathtt{B} < 0 \wedge \mathtt{A} \leq 0) \vee (\mathtt{A} \geq 1 \wedge \mathtt{A} > \mathtt{B}) \\ \varphi_s = (\mathtt{B} \geq 0 \wedge \mathtt{A} \leq 0) \vee (\mathtt{A} \geq 1 \wedge \mathtt{B} \geq \mathtt{A}) \end{array} . \tag{2}$$

3. Trace Elimination (TE). TE refines a program P by eliminating a set of AND-trees from P while preserving the rest of its AND-trees. While the elimination of infeasible trees

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

```
wh_2_q(A, B) ← B < 0, A ≤ 0.
wh_1_q(A, B) ← D < 0, C = 0, A = 1, B = D + 1, wh_2_q(C, D).
wh_1_q(A, B) ← C ≥ 1, A = C + 1, B = D + 1, wh_1_q(C, D).
```

Fig. 5. Excerpt of query clauses corresponding to the program in Figure 3 (left) starting from the goal (query) error; the suffix q denotes a query predicate.

does not have any effect on preconditions, extra care must be taken while eliminating feasible ones. Lemma 3.2 allows us to derive a safe precondition in this case.

Definition 7 (θ_t)
Let P be a program and t a *feasible* AND-tree derived from P for exit0 or error. Let $p^I(\mathbf{x})$ be the atom label of an initial node of t. Then $\theta_t = \mathsf{constr}(t)|_{\mathbf{x}}$ is a necessary condition for t to be feasible.

Lemma 3.2 (*Adapted from Kafle et al. (2018) for AND-tree of* exit0)
Let P' be the result of eliminating a *feasible* AND-tree t for exit0 (resp. error) from P. Then $\mathsf{nec_s}(P) = \mathsf{nec_s}(P') \vee \theta_t$ (resp. $\mathsf{nec_u}(P) = \mathsf{nec_u}(P') \vee \theta_t$), where θ_t is a constraint extracted from t (Definition 7).

Observe that the elimination of feasible traces acts as program decomposition. Transformations such as PE, CS and TE (when used to remove *infeasible* trees) not only preserve the goal but also the initial clauses. This allows us to construct a sequence of clauses $P_0, P_1, \ldots, P_m$ where $P = P_0$ and each element of the sequence is more specialized than its predecessor wrt. derivations of exit0 (error). As a consequence, the NPs are more precise. We write $P \Longrightarrow_A P'$ when P' is a goal-preserving transformation of P wrt. an atom A, that is, $P \models A$ iff $P' \models A$. TE (eliminating *feasible* trees) is a little different, in that it does not preserve the goal. We abuse the notation and write $P \Longrightarrow_{t_A} P'$ for transformation of P by eliminating a feasible tree rooted at A, yielding P'. Lemma 3.2 ensures soundness of preconditions in this case.

Let us now wrap these transformations and their combinations. Let tr and tr-seq be any functions satisfying the following:

$$\mathsf{tr}_A\langle P, \varphi\rangle = \begin{cases} \langle P', \varphi\rangle & \text{where } P \Longrightarrow_A P' \\ \textit{or} & \\ \langle P', \varphi'\rangle & \text{where } P \Longrightarrow_{t_A} P' \text{ and } \varphi' = \varphi \vee \theta_{t_A} \text{(Definition 7)} \end{cases}$$

$$\mathsf{tr\text{-}seq}_A\langle P, \varphi\rangle = \mathsf{tr}_A^n\langle P, \varphi\rangle \text{ for } n \geq 1$$

where $f^n = f^{n-1} \circ f$, with $f^1 = f$. tr-seq allows us to combine the above transformations in any order and Proposition 3.3 allows us to derive more precise preconditions.

Proposition 3.3
Let P be a program, $\langle P_s, \varphi_s\rangle = \mathsf{tr\text{-}seq}_{\mathsf{exit0}}\langle P, \mathit{false}\rangle$. Then $\models_{\mathbb{T}} (\mathsf{nec_s}(P_s) \vee \varphi_s) \rightarrow \mathsf{nec_s}(P)$. Similarly, if $\langle P_u, \varphi_u\rangle = \mathsf{tr\text{-}seq}_{\mathsf{error}}\langle P, \mathit{false}\rangle$, then $\models_{\mathbb{T}} (\mathsf{nec_u}(P_u) \vee \varphi_u) \rightarrow \mathsf{nec_u}(P)$.

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

Algorithm 1 Inferring sufficient preconditions

1: **Input**: Program P with clauses for exit0, error and init; and trans. seq. tr-seq.
2: **Output:** Pair of SPs for the safety and unsafety of P (wrt. exit0 and error).
3: **Initialization:** $\psi_s \leftarrow \mathit{false}$; $\psi_u \leftarrow \mathit{false}$; $P_s \leftarrow P$; $P_u \leftarrow P$;
4: $\varphi_{old} \leftarrow \mathsf{nec_s}(P)$ (Definition 4); itr $\leftarrow 0$;
5: **while** true **do**
6: $\langle P_s, \theta_s\rangle \leftarrow \mathsf{tr\text{-}seq}_{\mathsf{exit0}}\langle P_s, \mathit{false}\rangle$ ‖ $\langle P_u, \theta_u\rangle \leftarrow \mathsf{tr\text{-}seq}_{\mathsf{error}}\langle P_u, \mathit{false}\rangle$
$\varphi_s \leftarrow \mathsf{nec_s}(P_s) \vee \theta_s$ ‖ $\varphi_u \leftarrow \mathsf{nec_u}(P_u) \vee \theta_u$
7: $\varphi_{new} \leftarrow \varphi_s \wedge \varphi_u$
8: **if** $\varphi_{new} \equiv \mathit{false}$ **then** ▷ *separating* condition reached
9: $\psi_s \leftarrow \psi_s \vee \varphi_s$ ‖ $\psi_u \leftarrow \psi_u \vee \varphi_u$
10: **return** $\langle \psi_s, \psi_u\rangle$
11: **if** $\varphi_{old} \models_{\mathbb{T}} \varphi_{new}$ **then** ▷ approximation was not strengthened
12: $\psi_s \leftarrow \psi_s \vee (\varphi_s \wedge \neg\varphi_u)$ ‖ $\psi_u \leftarrow \psi_u \vee (\varphi_u \wedge \neg\varphi_s)$
13: **return** $\langle \psi_s, \psi_u\rangle$
▷ refine programs by constraining initial clauses with φ_{new} (Definition 3)
14: $\psi_s \leftarrow \psi_s \vee (\varphi_s \wedge \neg\varphi_u)$ ‖ $\psi_u \leftarrow \psi_u \vee (\varphi_u \wedge \neg\varphi_s)$
$P_s \leftarrow P_{s\varphi_{new}}^{I}$ ‖ $P_u \leftarrow P_{u\varphi_{new}}^{I}$
15: $\varphi_{old} \leftarrow \varphi_{new}$; itr $++$;

4 An algorithm for precondition inference

We now give an algorithm for computing SPs for safety and unsafety as Algorithm 1 based on the transformations described previously. Input is a set of CHCs (involving clauses for exit0, error and init) and a sequence of transformations tr-seq. Output is a pair of SPs for safety and unsafety. The SPs ψ_s and ψ_u are initialized to *false*. The algorithm aims to weaken these SPs as far as it can. φ_{old} keeps track of the set of initial states that are yet to be proven safe or unsafe. P_s and P_u respectively keep track of the transformations of P with respect to exit0 and error.

In the algorithm, the following operations are carried out in an iterative manner and possibly in parallel (within the *while* loop). The instructions on two sides of the boxes can be executed in parallel. One or more of the transformation of P_s and P_u with respect to exit0 and error, respectively, are carried out and the NPs are extracted from the resulting programs (*line 6*). The algorithm terminates and returns an SP if the conjunction of these NPs is unsatisfiable (*line 10*, *separating*) or it is not stronger (wrt. $\models_{\mathbb{T}}$) than φ_{old} (*line 13*). Otherwise, the algorithm iterates with revised programs obtained by constraining their initial clauses with the conjunction φ_{new} (*line 14*). For this, φ_{new} needs to be converted to DNF that may blow up the number of resulting *initial clauses*. In our experiments, the largest size of DNF discovered was 11. Even if all transformations in the algorithm terminate, it may still not terminate since φ_{new} can infinitely be decreased. But it makes a progress, that is, it explores a strictly smaller set of initial states in each iteration that have not yet been known safe or unsafe. This is formalized in Proposition 4.1. Observe that each iteration computes valid SPs for the safety and unsafety of the original program (e.g. *line 14*) and combines them disjunctively with the previous SPs. Proposition 4.2 ensures that the combination yields valid SPs for the original program.

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

```
error ← A < 0, B = 0, wh_2(B, A).
wh_2(A, B) ← B < 0, A = 0, C = 1,
             B − D = −1, wh_1(C, D).
wh_1(A, B) ← A > B, B ≥ 0, A ≥ 1, init_1(A, B).
wh_1(A, B) ← A > B, A ≥ 1, A − C = −1,
             B − D = −1, wh_1(C, D).
init_1(A, B) ← A > B, B ≥ 0.
```

```
exit0 ← A ≥ 0, B = 0, wh_2(B, A).
wh_2(A, B) ← B ≥ 0, A = 0, C = 1,
             B − D = −1, wh_1(C, D).
wh_1(A, B) ← B ≥ A, A ≥ 1, init_1(A, B).
wh_1(A, B) ← B ≥ A, A − C = −1,
             A ≥ 1, B − D = −1, wh_1(C, D).
init_1(A, B) ← B ≥ A, A ≥ 1.
```

Fig. 6. Constraint specialized programs: wrt. error (left) and wrt. exit0 (right).

Proposition 4.1 (*Progress and Termination of Algorithm 1*)
Algorithm 1 either progresses or terminates.

Proposition 4.2 (*Composing Preconditions*)
Let Φ be a set of formulas such that each $\varphi \in \Phi$ is an SP for (un)safety of P. Then $\bigvee \Phi$ is also an SP for (un)safety of P.

Proposition 3.3 ensures the correctness of the transformations sequence, Proposition 3.1 ensures that the precondition of P^I_φ is also that of P, and Proposition 4.2 allows us to combine the preconditions derived in the separate iterations. Together they ensure the soundness of Algorithm 1:

Theorem 4.3 (*Soundness of Algorithm 1*)
Let P be a program annotated with the predicates exit0 (set of safe terminating states), error (set of unsafe terminating states) and init (set of initial states). If Algorithm 1 returns a tuple $\langle S, U \rangle$, then S and U are the SPs for safety and unsafety of P, respectively, with respect to the predicates exit0, error and init.

Impact of transformation sequence on preconditions. Let us apply the algorithm to the program P in Figure 1. Initially, $\varphi_{old} = \mathit{true}$, the initial state of P. First, we choose to apply PE wrt. error and wrt. exit0, obtaining the set of CHCs shown in Figure 3. The corresponding NPs are given in equation (1), from which $\varphi_{new} = \varphi_s \wedge \varphi_u \equiv \mathtt{B} \geq 0 \wedge \mathtt{A} \geq 1$. Since φ_{new} is *satisfiable*, the preconditions are not separating. Neither is condition $\varphi_{old} \models_{\mathbb{T}} \varphi_{new}$ satisfied, so the algorithm progresses to refinement (*line 14-16*). At this point, we compute SPs for both the safety and unsafety as below.

$$\psi_u = \mathit{false} \vee ((\mathtt{B} < 0 \vee \mathtt{A} \geq 1) \wedge \neg(\mathtt{B} \geq 0)) \equiv \mathtt{B} < 0$$
$$\psi_s = \mathit{false} \vee (\mathtt{B} \geq 0 \wedge \neg(\mathtt{B} < 0 \vee \mathtt{A} \geq 1)) \equiv \mathtt{B} \geq 0 \wedge \mathtt{A} \leq 0.$$

As the next step, we refine P to $P^I_{\varphi_{new}}$, in which the clause for `init_1` in Figure 3 gets strengthened to $\mathtt{init_1}(\mathtt{A}, \mathtt{B}) \leftarrow \mathtt{A} \geq 1, \mathtt{B} \geq 0$. The clause for `init_2` gets eliminated due to an unsatisfiable constraint in its body. The refined programs are trivial and are omitted. In the next iteration, we apply CS with respect to error and exit0, respectively, obtaining the clauses shown in Figure 6. Note that the clause $\mathtt{wh_2}(\mathtt{A}, \mathtt{B}) \leftarrow \mathtt{B} < 0, \mathtt{A} \leq 0, \mathtt{init_2}(\mathtt{A}, \mathtt{B})$ is removed since it is no longer feasible without the initial clause for `init_2`.

From these we derive: $\varphi_u = (\mathtt{B} \geq 0 \wedge \mathtt{A} > \mathtt{B})$ and $\varphi_s = (\mathtt{B} \geq \mathtt{A} \wedge \mathtt{A} \geq 1)$. Since we now have $\varphi_u \wedge \varphi_s \equiv \mathit{false}$, the preconditions are separating and the algorithm terminates. The final SPs are derived as the disjunction of SPs over the iterations, as follows: $\psi_u = \mathtt{B} < 0 \vee (\mathtt{B} \geq 0 \wedge \mathtt{A} > \mathtt{B})$ and $\psi_s = (\mathtt{B} \geq 0 \wedge \mathtt{A} \leq 0) \vee (\mathtt{B} \geq \mathtt{A} \wedge \mathtt{A} \geq 1)$.

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

If instead we apply CS∘PE to the original program at *line 6* (rather than applying single transformation in each iteration), we obtain *separating* preconditions in a single iteration as shown in equation (2), where $\varphi_s \wedge \varphi_u$ is *unsatisfiable*. This suggests that a well chosen transformation sequence may reduce refinement iterations and also avoid the costly DNF conversion needed at *line 14*. Based on this, we fix tr-seq to be TE∘CS∘PE during the experiments. Our experience shows that CS is most effective when performed after PE which not only performs control flow refinement of the program but also brings polyvariant specialisation. TE on the other hand helps decompose problem in addition to splitting predicates. But since it is an expensive operation, we apply it at last.

Non-termination. The SPs derived by our method may include non-terminating inputs, that neither lead to safe nor unsafe. Popeea and Chin (2013) treat such inputs as *unsafe* whereas Seghir and Schrammel (2014) ignore them, as do we. However, the modeling of safe and unsafe terminating states and their over-approximations allow us to reason about a limited form of non-termination as suggested by Popeea and Chin (2013): Any input state that is neither in the over-approximation of safe nor unsafe leads to non-termination assuming we model all terminating (un)safe states.

We demonstrate this with Figure 7. The program does not terminate if $a \in [0, 10]$. We derive $\varphi_u = \mathtt{a} < 0$ and $\varphi_s = \mathtt{a} \geq 11$ as NPs. Thus the condition satisfying $\neg(\varphi_u \vee \varphi_s)$, that is, $\mathtt{a} \in [0, 10]$ is a sufficient precondition for non-termination (which happens to be the exact condition in this case). It is obtained as a byproduct of our method; we leave the primary analysis of non-termination for future work.

```
void main(int a) {
  while (a ≥ 0) {
    if (a ≤ 9)  a++;
    else if (a == 10)
      a = 5;
    else return;
  }
  assert (false);
}
```

Fig. 7. Non-termination.

5 Experimental evaluation

Since we model both the safe and unsafe program states and successively refine them to be able to detect separating or more precise preconditions for the safety and unsafety of programs, the experiments were designed to *better* answer the following questions.

- Q1. Does the algorithm allow us to derive separating preconditions in practice?
- Q2. Does refinement allow us to derive more non-trivial or separating preconditions?
- Q3. How does our approach compare to that of state-of-the-art tools for precondition inference in terms of the quality of the preconditions and performance?

Experimental Setup. We implemented Algorithm 1 (a sequential version) in PI-Horn.[1] The implementation applies the sequence TE∘CS∘PE. The tool is written in Ciao Prolog (Hermenegildo *et al*. 2012), using PPL (Bagnara *et al*. 2008) and Yices2 (Dutertre 2014). Input is a set of CHCs, with exit0, error and init as distinguished predicates. PI-Horn outputs a pair of SPs for safety and unsafety and are classified as: (i) *optimal*: the

[1] "Precondition Inferrer for Horn clauses", available at https://github.com/bishoksan/PI-Horn.

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

Table 1. *Experimental results on 261 programs, with a timeout of 300 s*

iter	opt	ntS (Sw)	ntU (Uw)	ntSU	tSU		PI-Horn total/iter	WP-Rahft total/iter
0	58	0 (0)	0 (0)	0	0		58	197
1	87	9 (7)	5 (3)	2	20		99	20
2	21	20 (15)	7 (2)	5	0		43	0
3	5	6 (3)	3 (0)	3	0		11	0
4	2	3 (1)	2 (0)	2	0		5	0
5	2	0 (0)	0 (0)	0	0		2	1
6	1	0 (0)	0 (0)	0	0		1	0
#total	176	38 (26)	17 (5)	12	20		219	218

precondition is both necessary and sufficient (exact); (ii) *non-trivial*: the precondition is different from *false* (but not *optimal*) and (iii) *trivial*: the precondition is *false*.

Experiments were conducted on a MacBook Pro, running OSX 10.11 with 16GB memory and 2.7 GHz Intel Core i5 processor. We tested our approach with 261 integer programs (available from `https://github.com/bishoksan/PI-Horn/tree/master/benchmarks`) sourced as follows: (i) 150 integer programs from the *loop* (69) and *recursive* (81) subcategories of the *Integers and Control Flow* category of SV-COMP (Beyer 2021); (ii) 83 programs from the DAGGER (Gulavani *et al*. 2008) and TRACER tools (Jaffar *et al*. 2012) and (iii) 28 programs from the literature on precondition inference and backward analysis (Bakhirkin *et al*. 2014; Miné 2012; Moy 2008; Bakhirkin and Monniaux 2017; Cassez *et al*. 2017). We are unable to include some benchmarks used by Kafle *et al*. (2018) owing to unavailability of their C sources which are needed to model the exit states. Benchmark set (i) was designed for verification competitions, (ii) and (iii) to demonstrate particular tools and techniques. We adapt these C programs for precondition inference as follows. They are translated to CHCs of the required form based on specialization approach of De Angelis *et al*. (2017) using VeriMap (De Angelis *et al*. 2014). We then replace the generated $\mathsf{init}(\mathbf{x}) \leftarrow \varphi(\mathbf{x})$ clause by $\mathsf{init}(\mathbf{x}) \leftarrow \mathit{true}$. This allows analyses to infer preconditions in terms of $\mathbf{x}$, starting from an unrestricted set of *initial* clauses.

Results and Discussion. Table 1 shows the results. The columns 2–7 show results for PI-Horn and the last column for WP-Rahft (Kafle *et al*. 2018). The first column iter indicates the number of refinement iterations for both. The columns show opt (# programs with separating preconditions), ntS (Sw) (# programs with non-trivial SPs for safety excluding separating cases, and, in parentheses, the difference with trivial SPs for unsafety), ntU (Uw) (same, for unsafety), ntSU (# programs with non-trivial SPs for both safety and unsafety), tSU (# programs with trivial SPs for both safety and unsafety), total/iter (# programs with non-trivial (either for safety or unsafety) plus separating SP per iteration), WP-Rahft total/iter (# programs with non-trivial SP per iteration for WP-Rahft). For example, the entry 9 (7) in column 3 indicates that there were 9 non-trivial SPs for safety, of which 7 had trivial SPs for unsafety. In other words, the number in the parentheses counts the pairs of the form $\langle \mathsf{ntS}, \mathsf{tU} \rangle$ where tU means trivial precondition for unsafety. Each row corresponds to an iteration and contains the number of instances solved in that iteration, excluding those solved in the previous.

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

The results answer Q1 and Q2 positively. PI-Horn infers non-trivial preconditions for 83% and optimal ones for 67% of the programs. Interestingly, it infers optimal preconditions for 58 programs owing to specialization transformations alone (see row #1, iter 0), whereas it infers non-trivial preconditions for 99 programs (of which 87 are optimal) after the first refinement. More non-trivial preconditions are derived when refinement progresses. This indicates that both the preprocessing and refinement significantly increase the number of optimal (non-trivial) cases. However, for 63 out of 261 programs, refinement did not progress toward optimality (that is, it did not further shrink the approximations of safe and unsafe states). We also observe that it timed out on 9% and failed to infer any meaningful preconditions for 8%.

As for Q3 we could not meaningfully compare our tool against the work of Seghir and Schrammel (2014), or Bakhirkin *et al.* (2014), in the first case because of tool issues (discovered together with the authors), and a lack of automation (confirmed by the authors via email) for the second. We do compare with WP-Rahft (Kafle *et al.* 2018), but note that, while some of the components of the tools are identical, the results are not directly comparable. The success of PI-Horn depends on its ability to refine both the approximations simultaneously unlike WP-Rahft. For example, we might obtain a tight bound φ for safe states but if the approximation of unsafe states is ψ such that $\varphi \models_{\mathbb{T}} \psi, \psi \not\equiv \mathit{true}$ then PI-Horn returns trivial SP for safety whereas WP-Rahft returns non-trivial. On the other hand, WP-Rahft cannot detect optimality and does not derive preconditions for unsafety. The two tools provide almost the same number of programs with non-trivial SPs (219 vs 218), but they differ in the quality (e.g. optimality) of preconditions. Since WP-Rahft cannot detect optimality, limited information about it can be obtained by checking $\mathsf{suf}_{\text{PI-Horn}} \models_{\mathbb{T}} \mathsf{suf}_{\text{WP-Rahft}}$ on all those instances that are known to be optimal, where suf_x represents the sufficient precondition derived by the tool x. From this, we report that WP-Rahft derives optimal preconditions for 58 programs (# of successful checks), far less than PI-Horn (176). Every refinement yields improvements for PI-Horn but refinement beyond the second yields negligible improvements for WP-Rahft. This affirms that focusing attention on the intersection of approximations of safe and unsafe states is a good refinement strategy and shows the benefit of concurrently approximating these states. Thanks to the refined termination criteria of PI-Horn that the average time in seconds per instance is 30.72 (22 timeouts), while for WP-Rahft it goes from 28.14 (iter 1, 20 timeouts) to 41.86 (iter 6, 30 timeouts). In summary for Q3, PI-Horn infers *better* preconditions than WP-Rahft and shows reasonable performance.

6 Related work

Over-approximation techniques (forward/backward abstract interpretations or their combination (Cousot and Cousot 1992; Cousot *et al.* 2013; Bakhirkin and Monniaux 2017)) inherently derive NPs, and complementation supplies SPs at a cost of precision (due to approximation of the complement). Howe *et al.* (2004) use a pseudo-complemented domain (*Pos*) domain (Marriott and Søndergaard 1993) to infer SPs; Bakhirkin *et al.* (2014) exchange an abstract complement operation for abstract logical subtraction. Our method neither assumes an abstract domain is (pseudo-) complemented nor apply complementation of abstract elements during analysis. It applies to any abstract domain, and

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

complementation is carried out externally to abstract interpretation, storing the result as a formula without any loss of precision.

Little work has been done that inherently computes SPs without complementation. The notable exception is the work by Miné (2012), who designs all required purpose-built backward transfer functions for intervals, octagons and convex polyhedra domains. The downside is that the purpose-built operations, including widening, can be rather intricate and require substantial implementation effort. Moy (2008) employs weakest precondition reasoning and forward abstract interpretation to generalize conditions at loop heads to infer SPs. The derived conditions offer limited use except for a theorem prover. Our method, on the other hand, uses standard techniques and off-the-shelf tools. Output from PI-Horn can be consumed by other analysis and verification tools.

In a verification context, the dual-analysis approach of Popeea and Chin (2013) uses over-approximations, as we do, to concurrently infer NPs for safety and unsafety; from that, SPs are derived. No attempt is made to weaken those preconditions (by refining the approximations or focusing the analysis, as we do); we suspect such SPs are overly strong. Dillig *et al.* (2013) use Hoare style reasoning with abduction iteratively, to infer loop invariants that are sufficient to show validity of assertions. The success of their method relies on guessing good abducibles whereas computing precise inductive invariants is too hard to achieve for realistic programs (due to undecidability).

Program transformation approaches that preserve the goal can be used to derive preconditions, as our approach. These include the forward/backward iterative specialization by De Angelis *et al.* (2014), for verifying program properties. The transformation approach uses a constraint generalization instead of abstract interpretation. Similarly, the multivariant top-down analyzer by Puebla and Hermenegildo (1999); Muthukumar and Hermenegildo (1990) produces polyvariant specializations, as in the classical algorithms, performing backward analysis using abstract interpretation. These methods are complementary to ours and we leave a comparative study with our method for future work.

Seghir and Schrammel (2014) use a CEGAR approach to derive *exact* necessary and sufficient preconditions for safety. Like us, they model safe and unsafe states of a program and refine their approximations until they are disjoint. Their algorithm may diverge due to (i) the lack of a suitable generalization of the counterexamples (an inherent limitation of CEGAR) and (ii) the termination condition (disjointness) that is too hard to achieve for realistic programs (due to undecidability). Padhi *et al.* (2016) attempt to derive optimal preconditions using machine learning approaches. The success of their approach relies on learning good heuristics to separate good runs from bad runs. We, in contrast, use abstract interpretation and program transformation, so each step of the algorithm terminates and a sound precondition can be derived from the resulting programs. Besides, optimality is not the end goal for us and it is a byproduct of precision refinement.

The work of Kafle *et al.* (2018) is orthogonal to those above, combining a range of established techniques such as abstract interpretation, CEGAR and program transformations in a profitable way. The iterative nature of their approach allows them to derive more precise preconditions for safety, however the termination criterion, the maximum number of iterations supplied by the user, is rather weak and cannot be used to optimality of the preconditions. The current work offers several advantages. We model both safe and unsafe states that enables us to detect optimality and also infer NP and SP for both safety and unsafety. In addition, it allows reasoning about a limited form of

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

non-termination and provides more refined termination criteria. Unlike many methods in the literature (Seghir and Schrammel 2014; Bakhirkin *et al*. 2014), our method can uniformly handle programs with procedures and recursive programs.

7 Concluding remarks

We have presented an iterative method for automatically deriving sufficient preconditions for the safety and unsafety of programs. It maintains over-approximations of the set of *safe* and *unsafe* initial states. Each iteration of the algorithm considers only states that are common to these approximations as they are yet to be classified as safe or unsafe. The method terminates when the common set of states is empty or it fails to shrink in successive iterations. In experiments, the method generated separating preconditions in 67% of test cases and solved problems which fail to resolve using only approximation of unsafe states (as done in previous work). Owing to over-approximation, the sufficient preconditions may include some non-terminating states, which hinders the derivation of optimal preconditions. Our method can only infer preconditions that are expressible as boolean combinations of (quantifier free) linear integer constraints and the prototype implementation mostly ignores simplification of preconditions, possibly leaving redundancies. In future work, we intend to augment our method with non-termination analysis, extend it to infer quantified preconditions and work on simplifying the preconditions.

Acknowledgments

We thank John Gallagher and three anonymous reviewers whose suggestions helped improve the paper. We are also grateful for help with the use of VeriMap for C to CHC translation, provided to us by Emanuele De Angelis. Bishoksan Kafle has been partially funded by the Spanish Ministry of Research, Science and Innovation, grant MICINN PID2019-108528RB-C21 *ProCode* and Madrid P2018/TCS-4339 *BLOQUES-CM*.

References

Bagnara, R., Hill, P. M. and Zaffanella, E. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming 72,* 1–2, 3–21.

Bakhirkin, A., Berdine, J. and Piterman, N. 2014. Backward analysis via over-approximate abstraction and under-approximate subtraction. In *SAS 2014*. LNCS, vol. 8723. Springer, 34–50.

Bakhirkin, A. and Monniaux, D. 2017. Combining forward and backward abstract interpretation of Horn clauses. In *SAS 2017*. LNCS, vol. 10422. Springer, 23–45.

Beyer, D. 2021. Software verification: 10th comparative evaluation (SV-COMP 2021). In *TACAS 2021*, J. F. Groote and K. G. Larsen, Eds. LNCS. Springer, 401–422.

Beyer, D., Henzinger, T. A., Majumdar, R. and Rybalchenko, A. 2007. Path invariants. In *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 300–309.

Cassez, F., Jensen, P. G. and Larsen, K. G. 2017. Refinement of trace abstraction for real-time programs. In *Reachability Problems*. LNCS, vol. 10506. Springer, 42–58.

Codish, M. and Søndergaard, H. 2002. Meta-circular abstract interpretation in Prolog. In *The Essence of Computation*, T. Mogensen *et al*., Eds. LNCS, vol. 2566. Springer, 109–134.

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

Cousot, P. and Cousot, R. 1992. Abstract interpretation and application to logic programs. *J. Logic Programming 13,* 2&3, 103–179.

Cousot, P., Cousot, R., Fähndrich, M. and Logozzo, F. 2013. Automatic inference of necessary preconditions. In *VMCAI'13*. LNCS, vol. 7737. Springer, 128–148.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *POPL*. ACM Press, 84–96.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2014. Program verification via iterated specialization. *Science of Computer Programming 95*, 149–175.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2014. VeriMAP: A tool for verifying programs through transformations. In *TACAS 2014*. LNCS, vol. 8413. Springer, 568–574.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2017. Semantics-based generation of verification conditions via program specialization. *Science of Computer Programming 147*, 78–108.

Dillig, I., Dillig, T., Li, B. and McMillan, K. L. 2013. Inductive invariant generation via abductive inference. In *OOPSLA 2013*. ACM, 443–456.

Doménech, J. J., Gallagher, J. P. and Genaim, S. 2019. Control-flow refinement by partial evaluation, and its application to termination and cost analysis. *Theory and Practice of Logic Programming 19,* 5–6, 990–1005.

Dutertre, B. 2014. Yices 2.2. In *CAV 2014*. LNCS, vol. 8559. Springer, 737–744.

Gallagher, J. P. 2019. Polyvariant program specialisation with property-based abstraction. In *VPT 2019*. EPTCS, vol. 299. 34–48.

Gallagher, J. P. and Lafave, L. 1996. Regular approximation of computation paths in logic and functional languages. In *Partial Evaluation*. LNCS, vol. 1110. Springer, 115–136.

Grebenshchikov, S., Lopes, N. P., Popeea, C. and Rybalchenko, A. 2012. Synthesizing software verifiers from proof rules. In *PLDI 2012*. ACM, 405–416.

Gulavani, B. S., Chakraborty, S., Nori, A. V. and Rajamani, S. K. 2008. Automatically refining abstract interpretations. In *TACAS 2008*. LNCS, vol. 4963. Springer, 443–458.

Gulwani, S., Jain, S. and Koskinen, E. 2009. Control-flow refinement and progress invariants for bound analysis. In *PLDI*. ACM, 375–385.

Gupta, A. and Rybalchenko, A. 2009. Invgen: An efficient invariant generator. In *CAV*, A. Bouajjani and O. Maler, Eds. LNCS, vol. 5643. Springer, 634–640.

Gurfinkel, A., Kahsai, T., Komuravelli, A. and Navas, J. A. 2015. The SeaHorn verification framework. In *CAV 2015*. LNCS, vol. 9206. Springer, 343–361.

Hermenegildo, M. V., Bueno, F., Carro, M., *et al*. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1-2, 219–252.

Howe, J. M., King, A. and Lu, L. 2004. Analysing logic programs by reasoning backwards. In *Program Development in Computational Logic*. LNCS, vol. 3049. Springer, 152–188.

Jaffar, J., Murali, V., Navas, J. A. and Santosa, A. E. 2012. TRACER: A symbolic execution tool for verification. In *CAV 2012*. LNCS, vol. 7358. Springer, 758–766.

Jones, N., Gomard, C. and Sestoft, P. 1993. *Partial Evaluation and Automatic Software Generation*. Prentice Hall.

Kafle, B. and Gallagher, J. P. 2017a. Constraint specialisation in Horn clause verification. *Science of Computer Programming 137*, 125–140.

Kafle, B. and Gallagher, J. P. 2017b. Horn clause verification with convex polyhedral abstraction and tree automata-based refinement. *Computer Languages, Systems and Structures 47*, 2–18.

Kafle, B., Gallagher, J. P., Gange, G., *et al*. 2018. An iterative approach to precondition inference using constrained Horn clauses. *Theory and Practice of Logic Programming 18*, 553–570.

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press

Karr, M. 1976. Affine relationships among variables of a program. *Acta Informatica 6*, 133–151.

Marriott, K. and Søndergaard, H. 1993. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems 2,* 1–4, 181–196.

Miné, A. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation 19,* 1, 31–100.

Miné, A. 2012. Inferring sufficient conditions with backward polyhedral under-approximations. *Electronic Notes in Theoretical Computer Science 287*, 89–100.

Moy, Y. 2008. Sufficient preconditions for modular assertion checking. In *VMCAI 2008*. LNCS, vol. 4905. Springer, 188–202.

Muthukumar, K. and Hermenegildo, M. 1990. Deriving a Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759. April.

Padhi, S., Sharma, R. and Millstein, T. D. 2016. Data-driven precondition inference with learned features. In *PLDI 2016*. ACM, 42–56.

Peralta, J. C., Gallagher, J. P. and Sağlam, H. 1998. Analysis of imperative programs through analysis of constraint logic programs. In *SAS 1998*. LNCS, vol. 1503. 246–261.

Popeea, C. and Chin, W. 2013. Dual analysis for proving safety and finding bugs. *Science of Computer Programming 78,* 4, 390–411.

Puebla, G. and Hermenegildo, M. V. 1999. Abstract multiple specialization and its application to program parallelization. *Journal of Logical Programming 41,* 2–3, 279–316.

Sankaranarayanan, S., Sipma, H. and Manna, Z. 2004. Non-linear loop invariant generation using Gröbner bases. In *POPL*. ACM, 318–329.

Seghir, M. N. and Schrammel, P. 2014. Necessary and sufficient preconditions via eager abstraction. In *APLAS 2014*. LNCS, vol. 8858. Springer, 236–254.

https://doi.org/10.1017/S1471068421000272 Published online by Cambridge University Press